Chapter 15: Object-Oriented Programming (OOP)

The object-oriented paradigm relies on **polymorphism**, **encapsulation** and **inheritance**:

Polymorphism (多型)	1. Being able to refer to **different derived-classes in the same way** 2. But getting the proper behavior of each derived-class.
Encapsulation (封裝)	1. Data hiding 2. Any form of hiding
Inheritance (繼承)	1. Having one class be a special kind of another class 2. Example: `RegularStudent` is a special kind of `Student`

15.2 Inheritance 繼承

Inheritance (繼承) is an important mechanism in OOP. It allows us to create a class without declaring all of its members from scratch.

The new class can inherit members from an existing class.

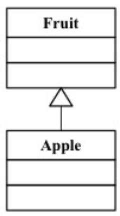
The existing class is called a superclass/parent-class/base-class (基礎類別), while the extended new class is called a subclass/child class/extended-class/derived-class (子類別).

Inheritance has three purposes:

1. The subclass may reuse its superclass by inheriting some members of the superclass, and then add new members or override some methods (functions) of the superclass.
2. Inheritance interprets "is-a" relationship. For example, if Fruit is the superclass, and Apple and Orange are subclasses of Fruit, the inheritance relationship between them states that "Apple is-a Fruit" and "Orange is-a Fruit".

   Common (共同) members between subclasses (Apple and Orange) appear in the superclass.
3. Polymorphism relies on the overriding mechanism of inheritance.

In Unified Modeling Language (UML), we use an empty triangle to represent the **is-a** relationship.



15.2.1 Basic Syntax of C++ Inheritance 繼承的基本語法

The base-class (e.g., `Fruit`) defines those members that are **common** to the types in the hierarchy.

Each derived-class (e.g., `Apple, Orange`) redefine (override) or define those members that are **specific** to the derived-class itself.

Base-class (基礎類別)

Suppose we like to model different kinds of pricing strategies for our bookstore.

- We'll define a class named `Quote` (報價), which will be the base-class. A `Quote` object will represent books in general.
- We will inherit a class `Bulk_quote` from `Quote` to represent specialized books that can be sold with a discount. (Bulk-buy 大批購買)

The `Quote` and `Bulk_quote` classes will have the following two member functions:

- `isbn()`, which will return the ISBN. This operation does not depend on the specifics of the inherited class(es); it will be defined only in class `Quote`.
- `net_price(size_t)`, which will return the price for purchasing a specified number of copies of a book. This operation is type specific; both `Quote` and `Bulk_quote` will define their own version of this function.

Quote.h

```
#ifndef QUOTE_H
#define QUOTE_H

#include <string>

class Quote {
public:
    Quote() = default;
    Quote(const std::string &book, double sales_price) :
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // returns total sales price for the specified number of items
    // derived-classes override for different discount calculation
    virtual double net_price(std::size_t n) const {
```

```
        return n * price;
    }
    virtual ~Quote() = default;// the destructor with override
private:
    std::string bookNo; // ISBN number of this item
protected:
    double price = 0.0; // normal, undiscounted price
};

#endif
```

Q: What's new?

A:

- The `protected` access label
- The use of the `virtual` keyword on the `net_price` function.
- The use of the `virtual` keyword on the destructor (explained later, but for now it is worth noting that base-class generally defines a virtual destructor.)

public, protected, private members

1. When designing a class to serve as a base-class, the criteria for designating a member as `public` do not change: interface functions (get and set for the data members) should be `public` and data generally should **not be** `public`.
2. A `private` member is prevented access from derived-classes. A `protected` member is allowed access from the derived-class.
3. A based class must decide which parts of the implementation to declare as `protected` and which should be `private`.
4. The interface to the derived type is the combination of both the `protected` and `public` members.

virtual member functions

Virtual functions are member functions whose behavior can be overridden in derived classes. The base-class defines **`virtual`** member functions if it expects its derived-classes to define the specific versions for themselves (the derived-class **override**s the virtual member function of the base-class). In our case, the `net_price` member function is type specific; both `Quote` and `Bulk_quote` will define their own version of this function. Thus, the `net_price` is a **`virtual`** member function and a keyword **`virtual`** is in place at the beginning of the function declaration.

```
class Quote {
public:
  std::string isbn() const;
  virtual double net_price(std::size_t n) const;
  ...
};
```

Example: we can use the base-class as usual.

```
#include "Quote.h"
#include <iostream>

using namespace std;

int main(){
    Quote q("032-171-4113", 42.38); // C++ Primer
    cout << "Net price for three copies of " << q.isbn()
      << " is: " << q.net_price(3) << endl;
    return 0;
}
```

```
Net price for three copies of 032-171-4113 is: 127.14
```

Derived-class (子類別)

Bulk_Quote.h

```
#ifndef BULKQUOTE_H
#define BULKQUOTE_H
#include "Quote.h"
#include <string>

class Bulk_quote : public Quote { // Bulk_quote inherits Quote
public:
  Bulk_quote() = default;
  Bulk_quote(const std::string& book, double p, std::size_t qty,
double disc): Quote(book, p), min_qty(qty), discount(disc) { }
  // overrides the base version
  double net_price(std::size_t) const override;
private:
  std::size_t min_qty = 0; // minimum purchase for discount
  double discount = 0.0; // fractional discount to apply
};
#endif
```

A derived-class must specify the class(es) from which it intends to inherit. It does so in a class derivation list, which is a colon followed by a comma-separated list of base-classes, each of which may have an optional access specifier:

```
class Bulk_quote : public Quote { // Bulk_quote inherits Quote
public:
    ...
};
```

Because `Bulk_quote` uses **`public`** in its derivation list, we can use objects of type `Bulk_quote` as if they were `Quote` objects (**public inheritance means is-a relationship**, more on this later). A `Bulk_quote` object is-a `Quote` object. (An `Apple` object is-a `Fruit`)

A derived-class must include in its own class body a declaration of all the `virtual` functions it intends to define for itself. A derived-class may include the `virtual` keyword on these functions but it is not required.

```
class Bulk_quote : public Quote { // Bulk_quote inherits Quote
public:
    double net_price(std::size_t) const;
...
};
```

Good Practice: A derived-class can explicitly note (compiling check) that its member function overrides a `virtual` member function that it inherits. It does so by specifying `override` after the parameter list, or after the `const` qualifier if the member is a `const` function.

```
class Bulk_quote : public Quote { // Bulk_quote inherits Quote
public:
    double net_price(std::size_t) const override;
...
};
```

Remarks:

1. Ordinarily, **derived-classes redefine the `virtual` functions that they inherit**, although they are not required to do so. If a derived-class does not redefine a `virtual`, then the version it uses is the one defined in its base-class.
2. When a derived-class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains `virtual` in all the derived-classes.
3. A derived-class function that overrides an inherited virtual function must have **exactly** the same parameter type(s) as the base-class function that it overrides. If the derived-class defines a function that has different parameters than the virtual function in the base-class, it overloads rather than overrides.

We are now ready to **redefine** `net_price` in `Bulk_quote`:

Bulk_Quote.cpp
```
#include "Bulk_Quote.h"

double Bulk_quote::net_price(size_t cnt) const {
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

Remarks:

1. A derived-class can access the `public` and `protected` members (e.g., `price`) of its base-class.
2. There is no distinction between how a member of the derived-class uses members defined in its own class (e.g., `min_qty` and `discount`) and how it uses members defined in its base-class (e.g., `price`).

15.2.2 Derived-class constructor and Constructor Chaining

The constructors of a base-class are **NOT** inherited by a derived-class, but each derived-class constructor explicitly or implicitly calls its base-class constructor, known as constructor chaining.

Explicitly: using the derived-class constructor initializer list to pass values to a base-class constructor.

```
Bulk_quote(const std::string& book, double p, std::size_t qty,
double disc) : Quote(book, p), min_qty(qty), discount(disc) { }
```

Implicitly: C++ automatically calls the base-class **default** constructor if we do not specify the base-class constructor.

Example: we are now ready to use the base-class and the derived-class.

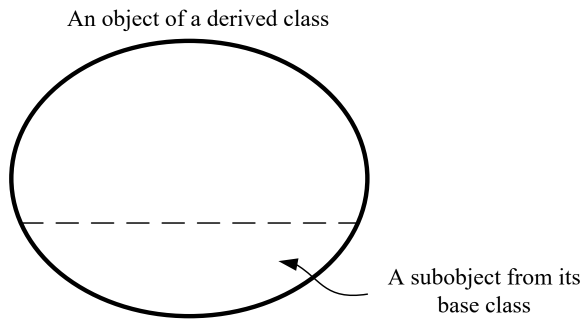
```
#include "Quote.h"
#include "Bulk_Quote.h"
#include <iostream>

using namespace std;
```

```
int main(){
    Quote q("032-171-4113", 42.38);
cout << "Net price for three copies of " << q.isbn() << " is: "
<< q.net_price(3) << endl;
    Bulk_quote bq("032-171-4113", 42.38, 10, 0.2);
cout << "Net price for three copies of " << bq.isbn() << " is: "
<< bq.net_price(3) << endl;
    cout << "Net price for 30 copies of " << bq.isbn() << " is: "
<< bq.net_price(30) << endl;
    return 0;
}
```

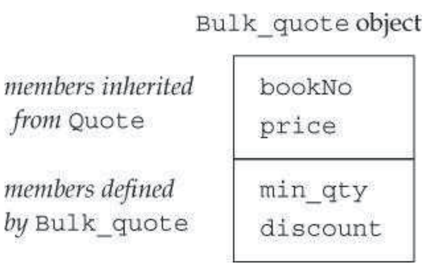
```
Net price for three copies of 032-171-4113 is: 127.14
Net price for three copies of 032-171-4113 is: 127.14
Net price for 30 copies of 032-171-4113 is: 1017.12
```

15.2.3 Object of the derived-class 子類別的物件

When you create an object of the derived-class, it contains within it **a sub-object of the base-class**.



Conceptually, we can think of a `Bulk_quote` object consisting of two parts as shown below:



This sub-object from its base-class is created by implicitly or explicitly calling the base-class constructor. If you think of a base-class as the parent to the derived-class the child, you know that **the base-class parts of an object have to be fully-formed before the derived-class parts can be constructed**.

15.2.4 Inheritance: an `is-a` Relationship and Dynamic Binding

Because a derived object contains a **subpart** corresponding to its base-class, we can use an object of a derived type **as if it were an object of its base type**. This is a very famous ***is-a*** relationship in OOP:

A derived object **is a** base object.

Ordinarily, C++ requires that references and pointer types match the assigned types, but this rule is relaxed for inheritance: we can bind a base-class reference or pointer to the base-class part of a derived object.

```
Quote item; // object of base type
Bulk_quote bulk; // object of derived type
Quote *p = &item; // p points to a Quote object
Quote *p2 = &bulk; // p2 points to the Quote part of bulk
Quote &r = bulk; // r bound to the Quote part of bulk
```

This derived-to-base conversion is implicit. It means that we can use **an object of derived type** or **a reference to a derived type** when **a reference to the base type is required**. Similarly, we can use a pointer to a derived type where a pointer to the base type is required.

Example

(IsAExample.cpp)

```
class Person{};
class Student : public Person{};

void dance(const Person& p){} // anyone can dance
void study(const Student& s){} // only students study

int main(){
    Person p;
    Student s;
    dance(p);
    dance(s);
    // study(p);  // COMPILING ERROR
    study(s);
    return 0;
}
```

Overriding vs. Overloading

Overriding means to provide a new implementation for a method in the derived-class and is a very **IMPORANT** mechanism in inheritance and polymorphism.

Overloading simply means to define multiple methods with the same name but different parameter lists. Overloading has nothing to do with inheritance.

<u>Overriding Example</u> (OverrideEx.cpp)

```
#include "iostream"
using namespace std;

class Base {
public:
    virtual void p(int i) {
        cout << "Base::p(int)" << endl;
    }
};

class Derived: public Base {
// This method overrides the method in Base
public:
    void p(int i) {
        cout << "Derived::p(int)" << endl;
    }
};

int main(){
    Base b;
    Derived d;
    Base* dp = new Derived();
    b.p(10);
    d.p(10);
    dp->p(10);
    delete dp;
    return 0;
}
```

Q: What is the output?

A:

```
Base::p(int)
Derived::p(int)
Derived::p(int)
```

<u>Possible unintentional mistake</u>: you are meant for overriding but unintentionally perform overloading:

OverrideExWrong1.cpp

```
#include "iostream"
using namespace std;

class Base {
public:
```

```
    virtual void p(int i) {
        cout << "Base::p(int)" << endl;
    }
};

class Derived: public Base {
// This method overrides the method in Base
public:
    void p(double i) {
        cout << "Derived::p(double)" << endl;
    }
};

int main(){
    Base b;
    Derived d;
    Base* dp = new Derived();
    b.p(10);
    d.p(10);
    dp->p(10);
    delete dp;
    return 0;
}
```

Q: What are the outputs?

A:

```
Base::p(int)
Derived::p(double) // not Base::p(int) but do a conversion from int to double[1]
Base::p(int)
```

To avoid this unintentional mistakes, you can specify `override` in the derived-class after the parameter list for compiling check.

<u>Possible unintentional mistake</u>: you forget to define `virtual` member function in the base-class. Again, this will unintentionally perform overloading:

OverrideExWrong2.cpp

```
#include "iostream"
using namespace std;

class Base {
public:
    virtual void p(int i) {
```

[1]Overridden functions are in different scopes (base-class v.s. derived-class); whereas overloaded functions are in the same scope (in derived-class only, or base-class only). URL: https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/
The exist of function `p` in the derived-class `Derived` will stop function name lookup in the base-class `Base` (function name `p` in the base-class `Base` is hidden from the derived-class `Derived`). Adding the line `using Base::p;` in the derived-class will allow that visibility, and overload will take place.

```
        cout << "Base::p(int)" << endl;
    }
};

class Derived: public Base {
// This method overrides the method in Base
public:
    void p(int i) {
        cout << "Derived::p(int)" << endl;
    }
};

int main(){
    Base b;
    Derived d;
    Base* dp = new Derived();
    b.p(10);
    d.p(10);
    dp->p(10);
    delete dp;
    return 0;
}
```

Q: What are the outputs?

A:

```
Base::p(int)
Derived::p(int)
Base::p(int)
```

15.3 Dynamic Binding 動態繫結

The aforementioned implicit derived-to-base conversion is the key behind **dynamic binding**. Through **dynamic binding**, we can use the same code to process objects of either type `Quote` or `Bulk_quote` **interchangeably**. For example, the following function prints the total price for purchasing the given number of copies of a given book:

```
double print_total(ostream& os, const Quote& item, size_t n){
  // depending on the type of the object bound to item
  // calls either Quote::net_price or Bulk_quote::net_price
  double ret = item.net_price(n);
  os << "ISBN: " << item.isbn() // calls Quote::isbn
    << " # sold: " << n << " total due: " << ret << endl;
  return ret;
}
```

This function is pretty simple—it prints the results of calling `isbn` and `net_price` on its parameter and returns the value calculated by the call to `net_price`. Because the `item` parameter is a **reference** to `Quote`, we can call this function on either a `Quote` object or a `Bulk_quote` object.

And because `net_price` is a `virtual` member function, the version of `net_price` that is executed will depend on the type of the object that we pass to `print_total`:

```
// basic has type Quote; bulk has type Bulk_quote
Quote basic;
Bulk_quote bulk;
print_total(cout, basic, 20); // calls Quote's net_price
print_total(cout, bulk, 20); // calls Bulk_quote's net_price
```

Example: we are now ready to put these lines of code together

UseDynamicBinding.cpp

```
#include "Quote.h"
#include "Bulk_Quote.h"
#include <iostream>

using namespace std;

double print_total(ostream& os, const Quote& item, size_t n){
    // depending on the type of the object bound to item
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() // calls Quote::isbn
       << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}

int main(){
    Quote basic("032-171-4113", 42.38);
    Bulk_quote bulk("032-171-4113", 42.38, 10, 0.2);
    Quote *pBulk = new Bulk_quote("978-0321714114", 42.38, 10,
0.2);
    print_total(cout, basic, 20); // calls Quote's net_price
    print_total(cout, bulk, 20); // calls Bulk_quote's net_price
    print_total(cout, *pBulk, 20); // calls Bulk_quote's net_price
    delete pBulk;
    return 0;
}
```

```
ISBN: 032-171-4113 # sold: 20 total due: 847.6
ISBN: 032-171-4113 # sold: 20 total due: 678.08
ISBN: 978-0321714114 # sold: 20 total due: 678.08
```

Remarks:

1. A reference or pointer of a base-class may refer or point to a base-class object or to an object of a derived-class.
2. This means that the type of object to which a reference or a pointer is bound may differ at run time (such as from user input). We call this **dynamic binding** or **late binding**.
3. Dynamic binding happens when a virtual member function is called through a reference (or a pointer) of a base-class. The run-time selection of `virtual` functions to run is relevant only when the function is called through a reference or a pointer.
4. If we call the `virtual` function on behalf of an object (as opposed through a reference or a pointer), then we know the exact type of the object at the compile time. The type is fixed and it does not vary at run time.

Another Example

```
#include <iostream>
#include <string>

class Energy {
public:
    virtual std::string print() {
        return std::string("Energy works");
    }
};

class Electricity: public Energy {
    std::string print() override {
        return std::string("Electricy works");
    }
};

class Heat: public Energy {
    std::string print() override {
        return std::string("Heat works");}

};

std::ostream& operator << (std::ostream& out, Energy* e){
    out << e->print();
    return out;
}

using namespace std;

int main () {
    Energy* e = new Electricity ;
    cout << e << endl ;
    Energy* h = new Heat ;
```

```
    cout << h << endl ;
    delete e;
    delete h;
    return 0 ;
}
```

```
Electricy works
Heat works
```